Temporal Deductive Verification of Basic ASM Models

HOCINE EL-HABIB DAHO[1]
DJILLALI BENHAMAMOUCH[2]

[1,2] Department of Computer Science
University of Oran, Algeria
[1]dahoh@yahoo.com

Abstract. Abstract State Machines (ASMs, for short) provide a practical new computational model which has been applied in the area of software engineering for systems design and analysis. However, reasoning about ASM models occurs, not within a formal deductive system, but basically in the classical informal proofs style of mathematics. Several formal verification approaches for proving correctness of ASM models have been investigated. In this paper we consider the use of the TLA$^+$logic for the deductive verification of a certain class of ASMs, namely *basic ASMs* which have successfully been applied in describing the dynamic behavior of systems at various levels of abstraction. In particular, we base our verification purpose on a translation of basic ASMs to the Temporal Logic of Actions (TLA) used as a formal basis to formally specify and reason about temporal behaviors of basic ASM models. The temporal deductive approach is illustrated by the formal correctness proof of a producer-consumer system formalized in terms of basic ASMs.

Keywords: Abstract State Machines (ASMs), Temporal Logic of Actions (TLA), Deductive Verification.

(Received December 21, 2009 / Accepted March 15, 2010)

1 Introduction

Abstract State Machines (ASMs), previously called *Evolving Algebras* and introduced by Y.Gurevich in [10], constitue the formal foundation of a practical methodology in modeling and analyzing various kinds of complex dynamic systems. The ASMs method has been successfully applied in different areas, such as software and hardware systems, programming languages, communication protocols and distributed algorithms (see [2, 3] for a comprehensive overview). It provides a flexible formalism to specify the operational semantics of a system at a natural abstraction level in a direct and intuitive way [3]. The ASM approach belong to the family of state-based methods, which model a system as a transition system. An ASM model describes the state space of a system by means of universes (i.e. basic sets) with functions and relations interpreted on them, and the state transitions by means of transition rules by which the system is driven from state to state. In ASMs, states are represented as first-order structures (*Algebras*) over the same signature (Vocabulary), and transition rules define the changes over time of the states. In applications, abstract state machines are considered a suitable specification formalism for giving semantics of a system in terms of its set of possible executions (i.e. state sequences).

Besides the standard mathematical techniques underlying the ASM approach that naturally support informal mathematical proofs of ASM model properties, there has been work on formal proof systems for ASMs, using various formal verification tools [1, 6, 7, 9, 13, 14, 15, 16]. For instance, [14] use the KIV (Karlsruhe Interactive Verifier) system to mechanically verify the proof of correctness of ASM refinements, both references [4] and [7] show how ASMs can be encoded in the PVS formal system in order to perform mechanical verification of the correctness of ASM specifications or to mechanically check hand proofs using the PVS proof system. While in [16, 15] algorithmic verification approaches based on model checking techniques have been applied for proving correctness of ASM specifications automatically.

In this paper, which is an extended version of our outlined work presented in [5], we propose to adopt Lamport's Temporal Logic of Actions (TLA) [11] as an appropriate alternative to the logic-based approaches [1, 6, 9, 7, 13, 15], to formally reason about basic ASM specifications (models) of dynamic systems. TLA is a state-based logic which provides the means for describing transition systems (i.e. states, state transitions and thereby the resulting state sequences) and formulating their properties in a single logical formalism, equipped with a relatively complete set of proof rules for reasoning about safety and liveness properties that can be required for systems. The operational behavior (semantics) of a basic ASM specification is directly defined by TLA-logical formulas and the TLA-proof techniques can be applied to formally prove the correctness of basic ASM specifications. Using this framework, both basic ASM specifications and required properties are represented by formulas in the same logic. In particular, we provide some basic rules to translate basic ASM models into TLA$^+$ specifications. TLA$^+$ is a formal specification language based on Zermelo-Fränkel set theory, first-order logic and the linear-time temporal logic TLA[12]. In addition to the operators of TLA, it contains operators for defining and manipulating data structures and syntactic structures for handling large specifications. The TLA$^+$ framework offers a potential mathematical logic framework into which ASM model elements are directly translated to their most natural equivalents in TLA$^+$.

The remainder of this paper is organized as follows: Section 2 we give a brief overview of related works. Section 3 briefly presents the basic notions of the ASM approach. In section 3, we give an overview of the main features of the TLA logic and the specification language TLA$^+$. Section 4 is mainly about the translation process of a basic ASM model into a TLA$^+$ specification. this section provides a description of how each element of a basic ASM model has to be encoded in TLA$^+$ expression. In section 5, we present a case study to illustrate the process of formal modelling and analysis of basic ASM model using the TLA$^+$-logical framework. Finally, in section 6, we conclude our contribution and outline futur research directions.

2 Related Work

Several attempts of applying formal verification techniques to ASM models have been investigated. In this section, we only report some deductive verification approaches for basic ASM models because they are the closest to our work.

In [13], Antje Nowack has suggested to use monodic fragments of First-Order Temporal Logic (FOTL) to verify ASMs. In particular, he proposed the definition of *guarded* ASMs and has shown that verification of the properties of such specific ASMs expressed in the guarded monodic fragment of FOTL is decidable. A reduction to the finite satisfiability problem of guarded monodic fragment is used. In this approach, the translation schemas address a restricted subset of the standard ASM language. The rules supported are *assignment* and *conditional* rules. Also, only functional symbols of arities at most one are supported. These are considerable limitations on the power and flexibility of the ASM specification language.

In more recent work [6], Fisher and Lisitsa present a method for verifying basic ASM model by translating them to an appropriate fragment of First-Order Temporal Logic(FOTL). The work presented is very much in spirit of the work done by Nowack. They have defined restrictions on basic ASM specifications (ASM-programs) which ensure that temporal translation falls into monodic fragment of FOTL, which is, in general, undecidable, but finitely axiomatizable. This allows them to use temporal translations for (semi-)automatic verifications of restricted basic ASM specifications either by decision procedures, or by theorem proving for restricted fragments of FOTL. Technically, Fisher's definition of *monodic* ASMs is less restrictive than that of *guarded* ASMs. On the other hand they can only guarantee existence of semi-decision procedure for *monodic* ASMs as opposed to decision procedure for *guarded* ASMs.

In the work on real-time systems by Beauquier and Slissenko [1], basic ASMs are represented by an extension of the theory of real addition and then the verification problem is discussed. The main open question in this work is to find a complete axiomatization for the proposed extension.

Work in [9], introduces a formal language for ASMs called FLEA, a system for formal reasoning about ASM behaviors. It has adopted a modal view, intended to catch the main ideas behind ASMs. FLEA has been extended to modal FLEA$^{\Box}$, for which a preliminary axiomatization is presented.

The work presented in this paper is similar to the work done by Fisher and Lisitsa, as our work presents a methodology for the deductive verification of basic ASM models based on a translation of basic ASMs to the Temporal Logic of Actions (TLA) as a formal basis. In contrast, our formalization approach to formally reason about basic ASM behaviors does not impose restrictions on the basic ASM specifications. Furthermore, the translation schemas support all basic ASM rules, as well as arbitrary n-ary functions. This is due to the expressive power and flexibility of the TLA$^+$ logical formalism equipped with a complete set of proof rules for reasoning about temporal properties that can be required for modeled systems in basic ASMs.

3 Basic Concepts of Abstract State Machines

Abstract State Machines (ASMs) [3] are used for modelling systems as transition systems. They define a state-based computational model, where computations(runs) are finite or infinite sequences of states $< S_i >_{i \geq 0}$, obtained from a given initial state S_0 by repeatedly executing transition rules. In ASMs, states are defined as many-sorted first-order structures over a given signature Σ (a vocabulary), and the transition relation is specified by transition rules for describing changes to states. States are implicitly given in an ASM model, and are usually described in terms of functions in the underlying signature. Abstract state machines are considered appropriate for giving semantics of a system in terms of its set of possible executions.

3.1 The Basic Model

An abstract state machine model, M, can be defined as a tuple of the form $M = \langle\ \Sigma,\ Prog\ ,\ Init\ \rangle$, where Σ is a signature, $Init$ is a closed formula over Σ describing the initial state and $Prog$ is a finite set of transition rules.

3.1.1 States :

States of M are variants of first-order structures over a given signature Σ. They are also called Σ-*Algebras*. A signature Σ consists of a collection of domain names (also called universe or set) and a collection of function names, each function name f coming with a fixed arity n and profile $T_1\times ...\times T_n \rightarrow T_k$ where $T_i(1 \leq i \leq n)$ and T_k are universe names (written $f : T_1\times ...\times T_n \rightarrow T_k$), or simply $f : \rightarrow T_k$ if $n = 0$.

A Σ-*Algebra* (or state) S consists of a nonempty set T^S for each universe T (the carrier set of T), and a function $f^S : T_1^S \times ... \times T_n^S \rightarrow T_k^S$ for each function name $f : T_1\times ...\times T_n \rightarrow T_k$ in Σ.

The universe names may be marked as *dynamic* or *static* according to whether or not the set of objects they contain may vary. Function names in Σ can be declared as :

- $Static$: static function names have the same fixed interpretation in each computation state; that is, static functions never change during a run.

- $Dynamic$: the interpretation of dynamic function names can be changed by the transitions occurring in a given computation step; that is, dynamic functions change during a run as a result of the specified system's behavior. Dynamic functions represent the internal state of the system.

Every ASM-signature Σ is assumed to contain the following logic symbols: static nullary functions $True$, $False$, $Undef$, the equality sign $=$, and the sort (universe) $Boolean$ with its usual boolean operators ($\neg$, $\wedge$, $\vee$, etc.)

3.1.2 Basic ASM Transition Rules :

Transition rules describe how transitions between states (*algebras* of signature Σ) can occur. They define the changes over time of the states of ASMs. The basic ASM transition rules are syntactic expressions generated as follows :

- $Skip\ rule$: the skip rule is the simplest transition rule. This rule causes no change to any function value. It is denoted as `Skip`

- $Update\ rule$: The update rule is an atomic transition rule, also called a local function update or simply update and has the form `c := t` where `c` is a variable (a dynamic nullary function in ASM terminology) and `t` is a closed term over Σ. The update `c := t` transforms the current state into a new state, in which the denotation of `c` has been changed into the current denotation of `t`.

  Likewise, if `f` is a dynamic function of arity `n`, and t_1, ..., t_n, t are terms over Σ, then `f(`t_1`, ....,`t_n`):= t` is also a function update. The effect of this update is that the denotation of `f` in the next state is equal to its denotation in the old state, except that the function value on the current values of t_1, ..., t_n is changed into the current value of t. This update also transforms the current state into a new state.

- $Conditional\ rule$: The conditional rule is a conditional update rule which specify a precondition for updating. It has the form `If` g `Then` R_T `Else` R_F where g, the guard, is a boolean expression and R_T , R_F are arbitrary update rules.

The `Else` part in a conditional rule may be omitted, implicitly making R_F = `Skip`. The meaning of this rule is that whenever the guard g evaluates to $true$ then apply R_T at the current state otherwise apply R_F. Note that the rules R_T and R_F can take the form of a block rule (see below).

- $Block\ rule$: the block rule has the form

```
block
  $R_1$
  $R_2$
  ....
  $R_n$
endblock
```

Where R_i for $1 \leq i \leq n$ are transition rules. The block rule groups a set of transition rules and fires them simultaneously. In ASMs, with the block rule we construct the overall ASM-program ($Prog$), and for brevity we always omit the keywords "block" and "endblock", and use indentation to eliminate ambiguity.

3.1.3 Computations :

A computation (run) of an ASM M is a finite (or infinite) sequence of states $\langle S_0, S_1, ..., S_k, ...\rangle$ such that S_0 refers to some given initial state and each state $S_{i+1} (i \geq 0)$ is obtained as the result of firing the program $Prog$ at S_i.

$$S_0 \overset{Prog}{\rightarrow} S_1 \overset{Prog}{\rightarrow} S_2 \overset{Prog}{\rightarrow} ... \overset{Prog}{\rightarrow} S_k \overset{Prog}{\rightarrow} ...$$

In this way, an ASM specification M which can be considered as given by a program $Prog$ together with an initial state S_0, models computations of dynamic systems through finite or infinite ASM runs.

4 Overview of TLA and TLA$^+$

The Temporal Logic of Actions (TLA) was proposed by Lamport [11] as a logic for specifying and reasoning about reactive, distributed, and particular asynchronous systems. TLA uses a single logical formalism for describing transition systems and formulating their properties. It is an extension of classical first-order logic by some linear-time temporal logic operators with a relatively complete set of proof rules. The semantics of TLA is defined in terms of states and behaviors. A state is an assignment of values to variables, and a behavior is an infinite sequence of states. A TLA formula is interpreted as a boolean function on behaviors.

In TLA we distinguish two classes of variables called *rigid variables* and *flexible variables*. The rigid variables represent quantities that do not change with time, they are also called constants. The flexible variables represent quantities that may change with time, and are just called variables.

TLA formulas are built up from *state functions* using the usual boolean operators ($\wedge$, $\vee$,$\neg$,$\Rightarrow$) and the operators $'$ (prime) and $\Box$ (read as always). A *state function* is defined as a non-boolean expression built upon variables and constant symbols. *State functions* are interpreted over single states. For example, the value of the *state function* $x + 1$ is 3 at a state s in which the value of x is 2. A *state predicate* is a boolean expression built from variables and constant symbols. For example, $x > 1$ is true in the state s. An *action* is a boolean-valued expression which can be made from variables, primed variables and constant symbols. *Actions* are interpreted over pairs of states. The unprimed variables are interpreted in the first state of a state pair and the primed variables in the second. For example, the $action$ $y' = x + 1$ is true over the pair of states $\langle s , t\rangle$ iff the value of y in t is equal to the value of x in s plus 1. A pair of states satisfying *action* A is called an A *step*. We write f' for the expression obtained by priming all the variables of the tuple (i.e. a *state function*) f , and $[A]_f$ for $(A \vee Unchanged\ f)$ where $Unchanged\ f \triangleq (f' = f)$, so an $[A]_f$ *step* is either an A *step* or a *step* that leaves f unchanged.

As usual in temporal logic, if F is a formula then $\Box F$ is true of a behavior iff it is true in every state of it. So, $\Box [A]_f$ holds over a behavior iff every pair of consecutive states in it is either an A *step* or a *step* that leaves f unchanged.

The standard way of specifying a system in TLA is with a formula in the "canonical form":
$Init \wedge \Box [Next]_f$, where

. $Init$ is the initial-state predicate, a formula describing all legal initial states of the system

. $Next$ is the next-state relation, which specifies all possible steps (pairs of successive states) in a behavior of the system. It is a description of actions that describe the different system operations.

. f consists of the variables the system operations can change.

TLA$^+$ is a formal specification language based on (untyped) Zermelo-Fraenkel set theory in which every value is a set, first-order logic, and TLA logic [12]. TLA$^+$ supplements TLA with operators for defining and manipulating data structures and mechanisms (syntactic structures) for writing specifications modularly. A TLA$^+$ specification is organized as a collection of modules. Logically, a TLA$^+$ module consists of a list of statements, where a statement can be a *declaration*, a *definition*, an *assumption* or a *theorem*. It has the following form:

```
MODULE ⟨Name⟩
CONSTANTS ⟨List of constant parameters⟩
VARIABLES ⟨List of variable parameters⟩
ASSUME ⟨Properties of constants⟩
TYPE INVARIANT ⟨Properties of variables⟩
INIT ⟨Initial values of variables⟩
SPEC ⟨A formula describing possible behaviors⟩
THEOREM ⟨A formula stating properties of spec⟩
END.
```

Below are some of the TLA$^+$ notations used to represent functions :

- The TLA$^+$ expression $[x \in S \mapsto e(x)]$ defines the function f whose domain is the set S such that $f\,[d] = e(d)$ for every $d \in S$.
- The TLA$^+$ notation $[A \rightarrow B]$ represents the set of all functions from the set A into the set B.
- If f is a function, the TLA$^+$expression [f EXCEPT![i] = j] defines a new function which is equal to f except in i where the returned value is j. So, the assertion $f' = [f$ EXCEPT![i] = $j]$ means $f'[i] = j \wedge \forall\, d \neq i : f'[d] = f[d]$.
- The function application is expressed using square brackets, so the notation $f\,[i]$ represents the value obtained from function f with argument i.

5 TLA$^+$Interpretation of Basic ASM Models

TLA$^+$ provides a unified logical framework for formalizing the execution semantics of ASM descriptions, and in which we are able to formally representing runs of ASM-programs in logical terms. In this section, we present a translation method for the formalization of basic ASM models. This defines a logical semantics which enables the deductive verification of systems modeled in ASMS. The generic translation rules show how each aspect of a basic ASM model has to be encoded into an equivalent TLA$^+$-expression. Through this translation process, we assume that the signature of a basic ASM model is thought of as being embedded in TLA$^+$-language. Hereafter we summarize the TLA$^+$translations of the different aspects of a basic ASM model.
An ASM model translates into a TLA$^+$module, a specification unit in the TLA$^+$language.

Sorts Representation : sorts have different TLA$^+$encoding depending on their being static or dynamic :

- A static sort U, i.e. a sort which does not change during computation, is translated as a TLA$^+$ constant parameter (*rigid variable*) U, declared in the TLA$^+$module with the CONSTANTS statement, as follows : **Constants** U. In TLA$^+$, we can give the definition of the constant value in the **ASSUME** statement, in terms of the TLA$^+$ build-in operators.
- A dynamic sort D, i.e. a sort which may change during computation, is encoded as a TLA$^+$variable parameter (*flexible variable*) D, declared with the VARIABLES statement, as follows: **Variables** D.

Functions Representation : ASM basic functions are classified in static functions which remain constant, and dynamic function which may change interpretation during computation.

- A static function f together with its signature (e.g. $f : S \rightarrow T$) that provides the type information of the function symbol f, is encoded as a TLA$^+$constant parameter f together with a set membership assumption (such as $f \in [S \rightarrow T]$) declared in the ASSUME statement of the TLA$^+$module, asserting that its value is a function in the set of all functions (with domain S and range a subset of T), described by the TLA$^+$construct $[S \rightarrow T]$. Furthermore, the definition of static function bodies can be given in the **ASSUME** statement.
- A dynamic function h together with its typing information, is encoded as a TLA$^+$variable parameter h. The type property of the variable h will be specified by a TLA$^+$ typing predicate namely a state predicate which has not to be assumed but to be proved as an invariant of the resulting TLA$^+$ temporal formula identifying behaviors of the basic ASM model.

Remark : Since First-Order Logic is a subset of both ASM language and TLA$^+$language, each first-order formula over an ASM-signature is transliterated to a TLA$^+$-formula with the same corresponding parameters. As a special case, the initial condition describing the starting state of an ASM run is transliterated to a TLA$^+$state formula specifying the initial values of ASM-variables (dynamic functions in ASM terminology).

TLA$^+$Translations of Basic ASM Rules : We now describe how ASM transition rules are translated into TLA$^+$ formulas (*actions*) which capture their execution effect upon a given state. In the following, we will consider translation schemas for the standard basic ASM rules, namely *Skip, Atomic update, Conditional, and Block* rules. To simplify notation, let consider R and R_i $(i \geq 1)$ stand for ASM rules, exp for an expression, g for a boolean expression(condition), x for a variable(dynamic nullary function), f for a dynamic function with arity $n > 0$, $V(R)$ for the set of variables potentially changed by R, V for the set of all declared ASM-program variables, and $[[R]]$ for the translation of R into TLA$^+$, namely the resulting TLA$^+$action which describes the execution semantics of the rule R. Notice that action $[[R]]$ does not state anything about those variables that are not used in R; where needed, such an action is obtained of $V(R)$ by the stuttering action as follows : $Unchanged(V - V(R))$.

- Skip rule of the form, R :: `Skip`, is a null statement(i.e., a no-op statement). It does nothing. In TLA$^+$ this expressed by an action formula that is identically $true$, where: $[[R]] \triangleq true$, and $V(R)$=$\emptyset$.

Basic update rules : The most basic forms of updates that can appear in an ASM model are called local function updates. They are similar to assignments in imperative programming languages :

- Updates of the form, R :: x `:=` exp. The effect of this update upon a state is to change the value of x into the value of exp. In TLA$^+$, the corresponding change to x is expressed by the TLA$^+$-*action* $[[R]]$ using the prime **′** operator as follows : $[[R]] \triangleq x' = exp$, where $V(R)$={x}.

- Updates of the form, R :: $f(t_1, ..., t_n)$`:=` exp, is represented by the TLA$^+$-*action* $[[R]]$ using the EXCEPT construct, the prime $'$ operator as follows: $[[R]] \triangleq f' = [f \text{ EXCEPT!}[t_1, ..., t_n] = exp]$, where $V(R)$={f}. The TLA$^+$-expression $[f$ EXCEPT!$[t_1, ..., t_n] = exp]$ represents the new function f' that is the same as f except that f' $[t_1, ..., t_n]$-$= exp$.

- **Conditional rule** of the form, R :: `If` g `Then` R_1 `Else` R_2, is the most common means of specifying a precondition for updating. Its TLA$^+$semantics is expressed by using the logical conditional choice operator, namely the IF $condition$ THEN $formula1$ ELSE $formula2$ structure which is equivalent to $((condition \wedge formula1) \vee (\neg condition \wedge formula2))$. The rule R is translated as follows :

$$
\begin{array}{l}
[[R]] \triangleq \wedge \text{ IF } g \text{ THEN} \\
\quad \wedge\ [[R_1]] \wedge Unchanged\ (V(R_2) - V(R_1)) \\
\quad \text{ELSE} \\
\quad\quad \wedge\ [[R_2]] \wedge Unchanged\ (V(R_1) - V(R_2)
\end{array}
$$

where $[[R_1]]$ and $[[R_2]]$ are the TLA$^+$translations of the sub-rules R_1 and R_2, and $V(R) = V(R_1) \cup V(R_2)$.

- **Block rule** of the form, R :: `Block` R_1 R_2 `...`R_n `endblock`, groups a set of sub-transition rules. With a block rule we construct the overall basic ASM-program namely $Prog$. Execution of a block rule amounts to the parallel execution of its sub-rules, provided these rules are pairwise consistent. TLA$^+$ provides different specification styles, namely *interleaving* and *noninterleaving* styles, for representing the concurrent execution of the sub-rules composing the block rule. With the block rule, we adopt the noninterleaving representation which allows simultaneity of transitions and simplifies rigorous reasoning about basic ASM-programs behaviours. According to the noninterleaving model of execution, the parallel execution semantics of the sub-rules within a block rule is formally represented by a TLA$^+$-based noninterleaving specification. The TLA$^+$-based noninterleaving representation associated with a block rule R is constructed from the TLA$^+$translations of the sub-rules $R_1, R_2, ..., R_n$ as follows:

$$
\begin{array}{l}
[[R]] \triangleq \vee\ [[R_1]] \wedge Unchanged\ (V(R) - V(R_1)) \\
\vee\ [[R_2]] \wedge Unchanged\ (V(R) - V(R_2)) \\
\vee \ldots \\
\vee\ [[R_n]] \wedge Unchanged\ (V(R) - V(R_n)) \\
\vee\ [[R_1]] \wedge [[R_2]] \wedge \\
\quad \wedge\ Unchanged V(R) - (V(R_1) \cup V(R_2))) \\
\vee \ldots \\
\vee\ [[R_1]] \wedge [[R_2]] \wedge [[R_3]] \wedge \\
\quad \wedge Unchanged V(R) - \bigcup_{i=1}^{3} V(R_i)) \\
\vee \ldots \\
\vee\ [[R_1]] \wedge [[R_2]] \wedge ... \wedge [[R_n]]
\end{array}
$$

Where $V(R) = V(R_1) \cup V(R_2) \cup ...\cup V(R_n)$, and $[[R_1]]$, ...,$[[R_n]]$ are TLA$^+$semantics of the sub-rules $R_1, R_2, ..., R_n$. The conjuncts $Unchanged$ (-

$V(R) - V(R_i))_{i \in 1..n}$ means that all variables which are not changed by the current rule R_i remain constant.

The TLA$^+$formula representing the temporal behaviors of a basic ASM model (specification)M is a safety formula Φ of the following form :

$$\Phi = [[M]] \triangleq INIT \wedge \Box\, [Next]_V$$

where $INIT$ is the state predicate representing the initial condition of the basic ASM model M, $Next$ is the TLA$^+$ action (semantics) for the basic ASM-program ($Prog$ namely the block rule) defined as a disjunction of actions capturing the possible individuals transitions and multiple simultaneous transitions, and V is the collection of ASM-program variables. The TLA$^+$ formula Φ describing an ASM model can be used to infer any safety property of an ASM system specification through logical reasoning. Formula $\Phi = [[M]]$ characterizes the set of all admissible behaviors, i.e. the behaviors of which it holds. The formula Φ produced in this way for an ASM model M captures the possible runs of this ASM.

6 Case Study : A Producer-Consumer System

In this section we use a case study to illustrate the process of formal modelling and analysis of a basic ASM model using the TLA$^+$-logical framework. The case study is a variant of distributed computing systems which may originate from quite different areas, including databases, communication protocols, namely a producer-consumer system, formalized in terms of basic ASM notation as presented in[8].

The setting is as follows : There are two processes (producer and consumer) which from time to time want to use the shared resource(the buffer) which must not be accessed by the two processes simultaneously. Distinguished items are produced, delivered(sent or deposited) to a buffer by a producer, later removed(received or accepted) from the buffer, and finally consumed by a consumer. The buffer is assumed to have capacity for one item.

The basic ASM model of the above informal description of the algorithm together with its temporal translation and correctness condition are given below:

6.1 ASM Specification :

The basic ASM model $M = \langle\, \Sigma, Prog\ , Init\, \rangle$ of the above system is based onto a signature including the 0-ary symbols x, y, buf. Their value may represent items to be processed by the system. Furthermore, the value of x and of y may be undefined (represented by x_undef and y_undef, respectively), and the buffer may be empty (represented by b_undef).

The ASM Signature: The signature Σ consists of the following universe and functions together with their associated sorts :

. Static universe : $Items$

. Static functions :

$$\begin{aligned} x\_undef &: \ \longrightarrow Items \\ y\_undef &: \ \longrightarrow Items \\ b\_empty &: \ \longrightarrow Items \end{aligned}$$

. Dynamic functions :

$$\begin{aligned} x &: \ \longrightarrow Items \\ item &: \ \longrightarrow Items \\ y &: \ \longrightarrow Items \\ buf &: \ \longrightarrow Items \end{aligned}$$

Initial State : $Init$ is a first-order formula specifying the initial state of any ASM run.
$Init \equiv x = x\_undef \wedge y = y\_undef \wedge buf = b\_empty$

Integrity Constraints : these are first-order descriptions which express implicit assumptions about the application domain.

$C \equiv x\_undef \neq y\_undef \neq b\_empty$

The ASM Program : Below is the set of transition rules describing behavior of the system. The processes interact as follows: In case the value of x is undefined, the value of $item$ is assigned to x, then forwarded to the empty buffer, removed from the buffer and assigned to y, and finally consumed.The basic ASM-program $Prog$ is as follows:

```
PROD::   If $x = x\_undef$ Then $x := item$

SEND::   If $\neg (x = x\_undef) \wedge buf = b\_empty$
           Then
            $buf := x$
            $x := x\_undef$

REC::    If $\neg (buf = b\_empty) \wedge y = y\_undef$
           Then
            $y := buf$
            $buf := b\_empty$

CONS::   If $\neg (y = y\_undef)$ Then $y := y\_undef$
```

6.2 TLA$^+$ Specification :

In the following we present the temporal translation of the ASM specification M of the producer-consumer system. The TLA$^+$module that encodes the basic ASM model M by following the translation schemas is as follows:

```
Module  ProducerConsumerASM

Constants  $Items,\ x\_undef,\ y\_undef,\ b\_empty$

Assume  $\wedge\ Items \neq \emptyset$
        $\wedge\ x\_undef, y\_undef, b\_empty \in Items$
        $\wedge\ x\_undef \neq y\_undef \neq b\_empty$

Variables  $x,\ y,\ buf,\ item$

$TypeInvariant \triangleq \wedge\ x \in Items$
                   $\wedge\ y \in Items$
                   $\wedge\ item \in Items$
                   $\wedge\ buf \in Items$

$V \triangleq \langle x, y, buf, item \rangle$

$INIT \triangleq \wedge\ x = x\_undef$
        $\wedge\ y = y\_undef$
        $\wedge\ buf = b\_empty$

$PROD \triangleq$ IF $x = x\_undef$ THEN $x' = item$

$SEND \triangleq$ IF $\neg (x = x\_undef) \wedge$
           $\wedge\ buf = b\_empty$
        THEN $\wedge\ buf' = x$
             $\wedge\ x' = x\_undef$

$REC \triangleq$ IF $\neg (buf = b\_empty) \wedge y = y\_undef$
        THEN $\wedge\ y' = buf$
             $\wedge\ buf' = b\_empty$

$CONS \triangleq$ IF $\neg (y = y\_undef)$ THEN $y' = y\_undef$

$NEXT \triangleq \vee\ PROD \wedge Unchanged(V - \{x\})$
 $\vee\ SEND \wedge Unchanged(V - \{buf, x\})$
 $\vee\ REC \wedge Unchanged(V - \{buf, y\})$
 $\vee\ CONS \wedge Unchanged(V - \{y\})$
 $\vee\ PROD \wedge REC \wedge Unchanged(V - \{x, y, buf\})$
 $\vee\ PROD \wedge CONS \wedge Unchanged(V - \{x, y\})$
 $\vee\ SEND \wedge CONS \wedge Unchanged(V - \{x, y, buf\})$

$\Phi = [[M]] \triangleq INIT \wedge \Box\, [NEXT]_V$

END.
```

6.3 Correctness Proof of the Basic ASM Model :

The purpose of giving a logical characterization of the behavior of an ASM specification using the TLA logic has been to be able to formally state and verify temporal properties using the proof rules of TLA. In TLA both ASM models and their required properties are represented in the same logic. The assertion " An ASM model M has the property P " is expressed in TLA by the validity of the formula $[[M]] \Rightarrow P$, where $[[M]]$ represents the TLA semantics of ASM system specification and $[[P]]$ is the logical expression of the informal property P.
As an example of correctness requirement that the producer-consumer system should satisfy is the exclusive sharing of buffer(mutual exclusion property), meaning that at any moment there can be at most one process accessing the buffer. This property of mutual exclusion for such a system is an example of invariance(safety) properties, those which are true at every state of the system execution. In TLA$^+$, this mutual exclusion property can be formally expressed by the temporal formula $\Box Mutex$, where $Mutex$ is a state predicate of the following form:

$$Mutex \triangleq \neg(x = buf \wedge y = buf)$$

The formal correctness proof of the ASM specification M which simulates the producer-consumer system behavior, is reduced to proving that the corresponding TLA$^+$-translation $[[M]]$ satisfies the mutual exclusion property $\Box Mutex$. In TLA$^+$, the assertion that the TLA$^+$-translation $[[M]]$ satisfies the property $\Box Mutex$ takes the form of the following theorem(a TLA formula) :

THEOREM: $[[M]] \Rightarrow \Box Mutex$

Which asserts that every behavior satisfying the TLA$^+$specification $[[M]]$ also satisfy the property $\Box Mutex$, i.e. the predicate $Mutex$ is true through every behavior satisfying this specification $[[M]]$.

6.3.1 Correctness proof :

The proof of $[[M]] \Rightarrow \Box Mutex$ is a relatively straightforward application of classical first-order reasoning and simple temporal facts which are embodied in the TLA proof rules [11]. In order to prove this property, we will take advantage of one of the rules of TLA, namely the rule $INV1$:

$$INV1 : \frac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box [N]_f \Rightarrow \Box I}$$

In the case of our specification, I is the state predicate $Mutex$ which is an invariant, N is the next-state action $NEXT$ and f is the tuple V of variables. The rule $INV1$ tells us that we must prove the following statements :

$$INIT \Rightarrow Mutex : (1)$$
$$Mutex \wedge [NEXT]_V \Rightarrow Mutex' : (2)$$

from which we can deduce $[[M]] \Rightarrow \Box Mutex$ as follows :

$$\begin{array}{ll} [[M]] \Rightarrow & \{ \text{By definition of } [[M]] \} \\ & INIT \wedge \Box\ [NEXT]_V \\ \Rightarrow & \{ \text{By the implication (1)} \} \\ & Mutex \wedge \Box [NEXT]_V \\ \Rightarrow & \{ \text{By the implication (2) and the rule INV1} \} \\ & \Box Mutex. \end{array}$$

To complete the proof, we must prove the statements (1) and (2), which involve simple predicate logic.

7 Conclusion and Further Work

In this paper we have presented a translation-based approach to formalize the operational semantics of basic ASM models in terms of TLA$^+$-logic. Though this formalization approach gives a temporal description of basic ASM models behaviors, it enables us to prove in a formal way correctness properties of systems specified in basic ASMs formalism. The proposed method has been illustrated by hand-translating the basic ASM specification of the producer-consumer system into a TLA$^+$specification. Moreover, we have shown how to formally prove required properties such as safety properties of basic ASM models by applying the deductive proof system of TLA. Futur development of this work includes the design and the implementation of a model translator, namely ASM2TLA$^+$ translator, to support the automatic translation of basic ASM specifications into TLA$^+$ specifications which can be verified automatically using the TLA$^+$ model checker called TLC[12]. On the other hand, we have in mind to extend the proposed approach to cover the specific cases of ASMs such as distributed ASMs, non-deterministic ASMs, and concurrent ASMs with the await construction.

References

[1] Beauquier,D.,Slissenko,A.: A first-order logic for specification of timed algorithms: Basics properties and a decidable class. Annals of Pure and Applied Logic, 113(1-3):13-52,2002.

[2] Börger,E.. High-level system Design and Analysis using Abstract State Machines. In Current Trends in Applied Formal Methods(FM-Trends98), number 1641 in LNCS, pages 1-43. Springer-Verlag, 1999.

[3] Börger,E., Stärk,R.: Abstract State Machines : A Method for High-Level System Design and Analysis. Springer 2003.

[4] Dol,A., Gaul,T., Vialard,U. and Zimmerman,w.: ASM-based mechanized verification of compiler back-ends. In Proceedings of the 28th Annual Conference of the German Society of Computer Science. Technical Report, Magdeburg University, 1998.

[5] El-Habib Daho,H., Benhamamouch,D.: Formal Verification of ASM Models Using TLA$^+$. In E.Börger et al.(Eds.), ABZ 2008, LNCS 5238, Springer-Verlag 2008.

[6] Fisher,M., Lisitsa,A.: Monodic ASMs and temporal verification. In Proceedings of Abstract State Machines Workshop ASM'2004, May 2004.

[7] Gargantini,A.,Riccoben,E.: Encoding Abstract State Machines in PVS. In Abstract State Machines : Theory and Applications, Vol.1912 of LNCS.Springer-Verlag, 2000.

[8] Glausch, A., Reisig, W.,: Distributed Abstract State Machines and Their Expressive Power. In D.Bjorner et al.(Eds.), *Logics of Specification Languages*, Springer-Verlag, 2008.

[9] Groenboom,R.,Lavalette,G.R.: A Formalisation of Evolving Algebras. In Proceedings Accolade 95, pages 17-28, 1995.

[10] Gurevich,Y.: Evolving Algebras : An attempt to discover semantics. Bulletin of the EATCS, (43):264-284, February 1991.

[11] Lamport,L.: The Temporal Logic of Actions. ACM Transactions on Programming Languages and systems, 16(3):872-923, May 1994.

[12] Lamport,L.: Specifying Systems : The TLA$^+$Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston 2003.

[13] Nowack,A.: Deciding the Verification Problem for Abstract State Machines. In Proceedings of ASM 2003, LNCS, 2589:341-371, 2003.

[14] Schellhorn,G.,Ahrendt,W.: Reasoning about Abstract State Machines : The WAM case study. Journal of Universal Computer science,3(4):377-413, 1997.

[15] Spielmann,M.: Automatic verification of Abstract State Machines. In Proceedings of 11th international conference on computer-Aided Verification(CAV'99), Vol.1633 of LNCS, pages 431-442. Springer-Verlag, 1999.

[16] Winter,K.: Model Checking Abstract State Machines. Journal of Universal Computer Science, 3(5):689-701, 1997.